World Academy of Science, Engineering and Technology
International Journal of Computer and Information Engineering
Vol:6, No:10, 2012

Optimizing Hadoop Block Placement Policy & Cluster Blocks Distribution

Nchimbi Edward Pius, Liu Qin, Fion Yang, Zhu Hong Ming

Abstract**—**The current Hadoop block placement policy do not fairly and evenly distributes replicas of blocks written to datanodes in a Hadoop cluster.

This paper presents a new solution that helps to keep the cluster in a balanced state while an HDFS client is writing data to a file in Hadoop cluster. The solution had been implemented, and test had been conducted to evaluate its contribution to Hadoop distributed file system.

It has been found that, the solution has lowered global execution time taken by Hadoop balancer to 22 percent. It also has been found that, Hadoop balancer respectively over replicate 1.75 and 3.3 percent of all re-distributed blocks in the modified and original Hadoop clusters.

The feature that keeps the cluster in a balanced state works as a core part to Hadoop system and not just as a utility like traditional balancer. This is one of the significant achievements and uniqueness of the solution developed during the course of this research work.

Keywords**—**Balancer, Datanode, Distributed file system, Hadoop, Replicas.

I. Introduction

HADOOP distributed data storage management and analytic framework is arguably the best large scale data processing solution available in today's distributed computing world. Although Hadoop system offers a wide range of solutions pertaining to distributed computing and huge data management problems, the system does not fairly and evenly distributes replicas of blocks across a cluster, such that once data are written to the cluster, a balancer has to be executed to keep the cluster in a balanced state**.**

When an HDFS client is writing data to a file, HDFS places the first replica on the node where the writer is located. The second and the third replicas are placed on two different nodes in a different rack. The rest are placed on random nodes with restrictions that no more than one replica is placed at any one node and no more than two replicas are placed in the same rack, if possible [1].

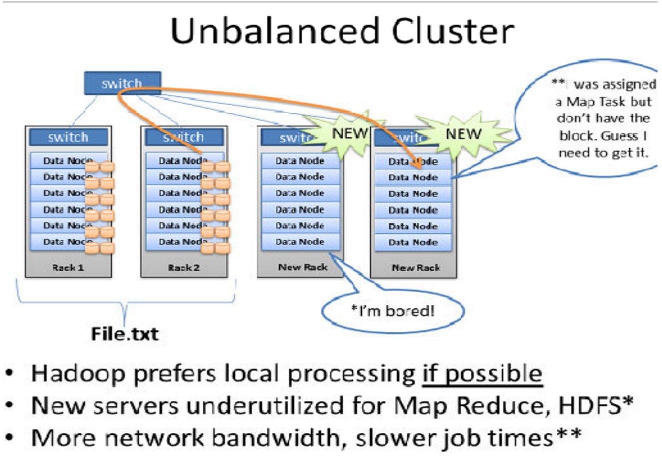
Over time the distribution of blocks across datanodes can become unbalanced. An unbalanced cluster can affect locality for MapReduce, and it puts a greater strain on the highly utilized datanodes, so it's best avoided [2]. Unfortunately the block placement strategy has not yet succeeded to put a fair and even distribution of blocks across the cluster. The HDFS block placement strategy does not take into account DataNode disk space utilization, therefore data might not always be placed fairly and uniformly to the DataNodes. Imbalance also occurs when new nodes are added to the cluster.

Nchimbi Edward Pius is pursuing a final academic year of Masters Degree in School of Software Engineering at School of Software Engineering, Tongji University, 201804, P.R China (phone: +86-150-007-285-70; fax: +86-21-695-898-40; e-mail: nchimbi2@yahoo.co.uk).

Prof. Liu Qin is Dean of School of Software Engineering with Tongji University, 201804, P.R China (e-mail: qin.liu@ tongji.edu.cn).

Fion Yang is Director for International students office with Tongji University, School of Software Engineering, Tongji University, 201804, P.R China (e-mail: fionyang2004@gmail.com).

Zhu Hong Ming is Lecturer with Tongji University, School of Software Engineering, Tongji University, 201804, P.R China (e-mail: hongming.zhu@ gmail.com).



- Hadoop prefers local processing if possible
- New servers underutilized for Map Reduce, HDFS*
- More network bandwidth, slower job times**

Fig. 1 Unbalanced Hadoop cluster

When we add new racks full of servers and network to an existing Hadoop cluster we can end up in a situation where the cluster is unbalanced. In this case (shown in Fig. 1), Racks 1 & 2 were our existing racks containing File.txt and running our Map Reduce jobs on that data. When we added two new racks to the cluster, our File.txt data did not automatically start spreading over to the new racks. All the data stays where it is [3]. The new servers were sitting idle with no data, until we start loading new data into the cluster. Furthermore, if the servers in Racks 1 & 2 are really busy, the Job Tracker may have no other choice but to assign Map tasks on File.txt to the new servers which have no local data. The new servers need to go grab the data over the network. As result you may see more network traffic and slower job completion times [3].

To overcome uneven block distribution scenario across the cluster, a utility program called balancer has to be explicitly executed by human being to re-distribute blocks in the cluster. The balancer is a tool that balances disk space usage on an HDFS cluster [1]. Balancer looks at the difference in available storage between nodes and attempts to provide balance to a certain threshold. New nodes with lots of free disk space will be detected and balancer can begin copying block data off nodes with less available space to the new nodes [3].

In a real world cluster like Amazon EC2, it may cost a lot of resource such as time and energy to run a balancer across such huge cluster to make it balanced. It would be better to have a solution that keeps track of and estimates utilization of a datanode prior to every attempt that puts a replica of block on the datanode. Some parts of the logic designed in this research work adapt approximate the same concept and impact of Hadoop balancer, but the effects take place in a dynamic, real-time and an incrementally fashion.

A. Proposed Solution

To keep a fair and an even block distribution across the cluster, a new logic that tracks and estimate utilization of both datanodes and cluster is needed. When a file read operation is in progress, the feature will evaluate utilization of a cluster, datanodes and threshold value (default to ten percent); thereafter the datanode whose utililization differs from that of cluster by lesser of threshold value will be selected as one of the best datanode onto which a block will be placed. Otherwise (if the datanode is over utilized) the logic discards it and proceeds to other datanodes.

B. Solution Flow Chart

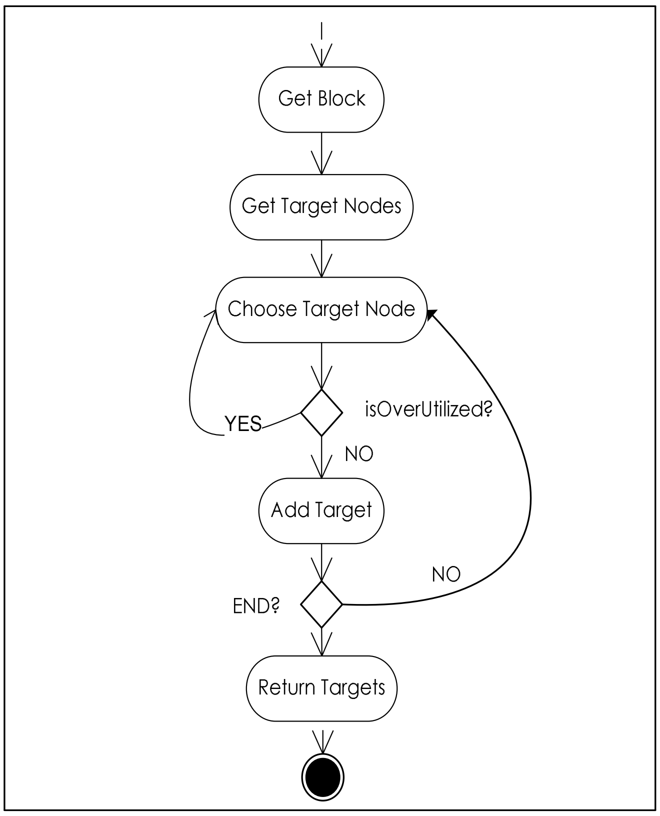


Fig. 2 Modified HDFS replication target chooser execution flow

C. Mathematical Analysis

This part analyses the proposed solution in a mathematical standpoint as follows:

Assumptions

- U_i and T_i are Used space and Total capacity of a datanode, respectively.
- β_i and β_c are Datanode and Cluster utilization, respectively.
- U_c and T_c are Used space and Total capacity of cluster, respectively.
- Δ is Threshold value.

Datanode and cluster utilization can respectively be expressed as follows:

$$\beta i = Ui \div Ti \qquad (1)$$
$$\beta c = Uc \div Tc \qquad (2)$$

A datanode is said to be over utilized if and only if the following inequality holds:

$$\beta i > \beta c + \Delta \qquad (3)$$

The proposed solution will only replicates blocks to the datanode whose utilization value (β_i) causes inequality (3) above to return false. This will ensure that a datanode do not get over utilized compared to other datanodes on a cluster.

D. Solution Class Diagram

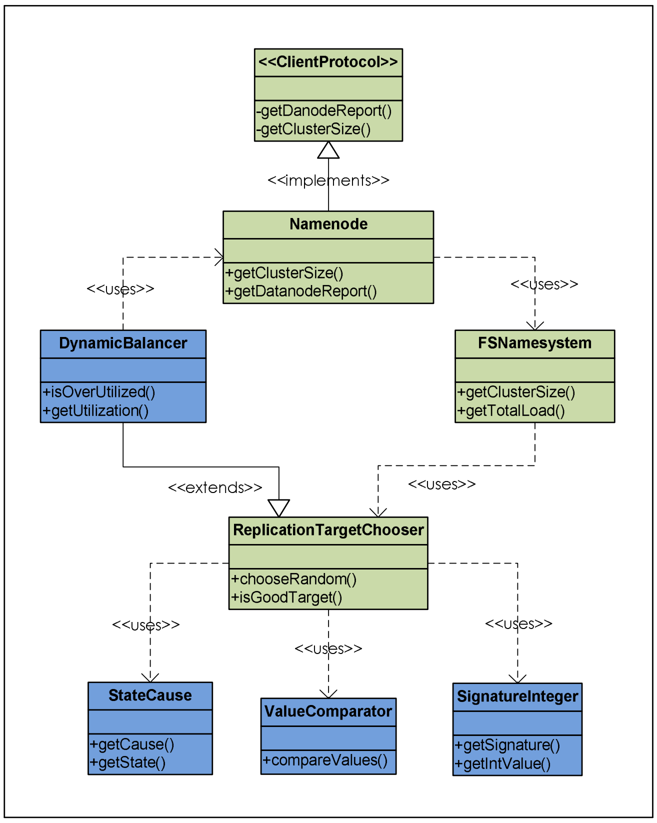


Fig. 3 Class diagram for some of additional features to existing Hadoop block placement policy

Classes that appear in blue in Fig. 3, represent new features that have been added to the existing Hadoop distributed file system. Those in green represent classes that previously existed but some methods and variables have been added onto them to accomplish the whole modified system business logic.

Some of the methods that have been added in both Namenode class and *ClientProtocol* interface include *getClusterSize()* and *getDatanodeReport()* which return size of cluster and datanodes available in a cluster respectively. The methods, *isOverUtilized()* and *getUtilization()* respectively return true if a datanode is over utilized and a double value that contains utilization of a datanode. Method *isGoodTarget()* returns a value whose type is of *StateCause*, a utility class.

The variables in *StateCause* utility class includes a *state* variable (either false or true) that indicates whether a datanode is bad(false) or good(true) target, and *cause* variable that defines a reason that causes the datanode to be good or bad target. These reasons include a datanode being in decommissioning, has less remaining space, is over utilized, has too much communication traffics and its rack's datanodes have been chosen for many times compared to other racks' datanodes in the cluster. The two helper classes *ValueComparator* and *SignatureInteger* have been internally used to compare datanodes utilization values and for identification purpose. Detailed information and functionality of all classes can be obtained in source code files (not included in this paper) that make up the whole system.

II. Experiment Design & Evaluation

A. Aim of Evaluation

- To verify that the modified HDFS system realizes the features defined in requirement section, in a practical standpoint.
- To collect and present blocks distribution data in each datanode across the cluster based on original and modified HDFS system respectively.
- To collect and present in a visual form used space in each datanode in a cluster based on original and modified HDFS systems respectively.
- To evaluate global time taken for a client to write a file in both original and modified HDFS system.
- To evaluate global time taken for a client to read a file in both original and modified HDFS system.

B. Experiment Setup

To achieve five evaluation objectives stated above, a cluster of three datanodes was setup as shown below. The cluster is made up of one designated namenode (that also serves as a datanode) identified by private IP address number 10.60.36.139 and two slaves datanodes whose private IP addresses are 10.60.36.140 and 10.60.36.77. User codes that write files to and read blocks of data from the cluster were all local to a datanode whose private IP address is 10.60.36.87 as it has been indicated in the Fig. 4.

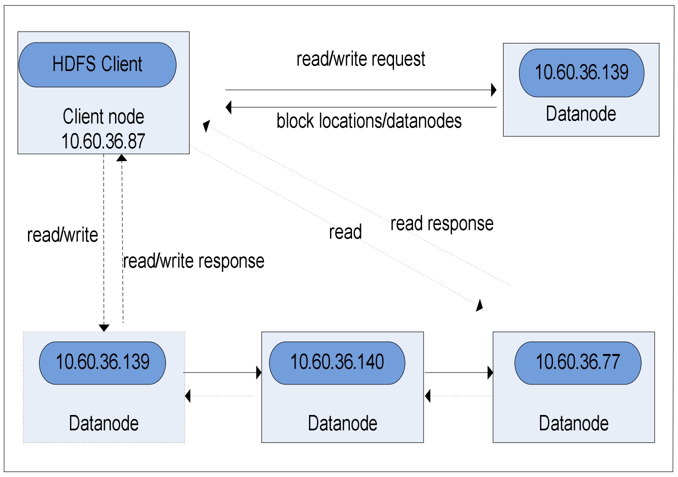


Fig. 4 A client writing files to and reading blocks of data from HDFS cluster

C. Evaluation Procedures

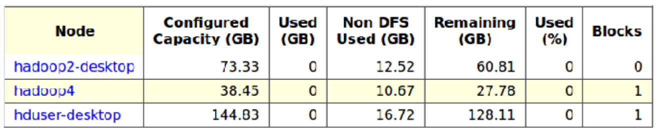
1. Install a modified version of HDFS system
2. Write files of different size to the cluster, by running a user program (implemented during the course of this research work) code that creates and writes data to a file.
3. Record time taken to write each file to the cluster
4. Record and visualize block distribution across the nodes that make up the cluster.
5. Record and visualize used space in each datanode that make up the cluster.
6. Uninstall the system installed in step 1 above.
7. Install an original (unmodified version) HDFS system. Repeat steps 2 through 5.

III. Results & Discussion

The data recorded for the two hadoop systems were visualized and presented in the following diagrams.

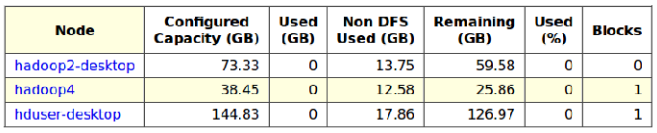
A. Cluster Initial View

Initially the two clusters' datanode R1:139 had a lowest capacity of 38.45 GB and datanode R1:77 had largest capacity of 144.83 GB. The replica factor of 2 was used throughout the course of this experiment. Hardware limitation (three nodes) was among the factors used to choose replication factor of two. In addition to that the value of 2 was chosen so as to observe behavior of the two systems when choosing two datanodes out of three datanodes onto which to replicate blocks of file. Detailed initial clusters' states can be seen from Figs. 5 and 6.



Node	Configured Capacity (GB)	Used (GB)	Non DFS Used (GB)	Remaining (GB)	Used (%)	Blocks
hadoop2-desktop	73.33	0	12.52	60.81	0	0
hadoop4	38.45	0	10.67	27.78	0	1
hduser-desktop	144.83	0	16.72	128.11	0	1

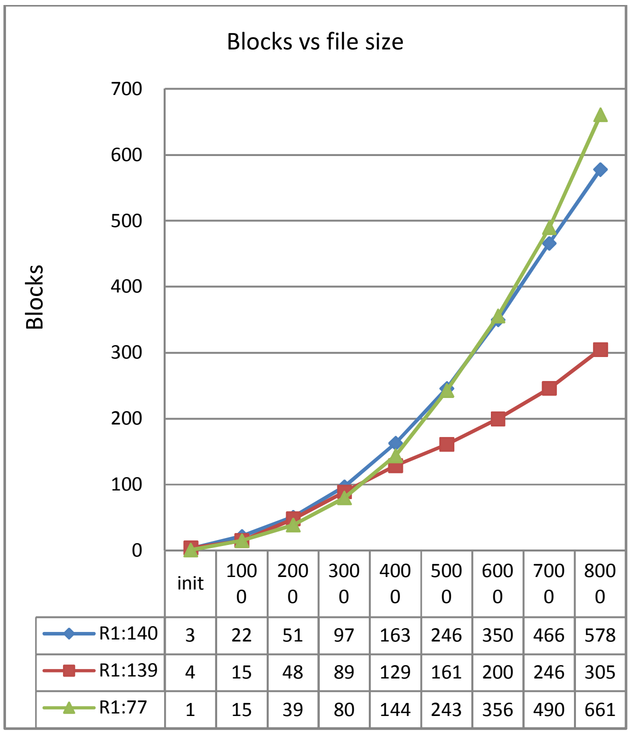
Fig. 5 Modified Hadoop system cluster's initial information



Node	Configured Capacity (GB)	Used (GB)	Non DFS Used (GB)	Remaining (GB)	Used (%)	Blocks
hadoop2-desktop	73.33	0	13.75	59.58	0	0
hadoop4	38.45	0	12.58	25.86	0	1
hduser-desktop	144.83	0	17.86	126.97	0	1

Fig. 6 Original Hadoop system cluster's initial information

B. Blocks versus File Size



	init	1000	2000	3000	4000	5000	6000	7000	8000
R1:140	3	22	51	97	163	246	350	466	578
R1:139	4	15	48	89	129	161	200	246	305
R1:77	1	15	39	80	144	243	356	490	661

Fig. 7 Block distribution across the modified Hadoop cluster

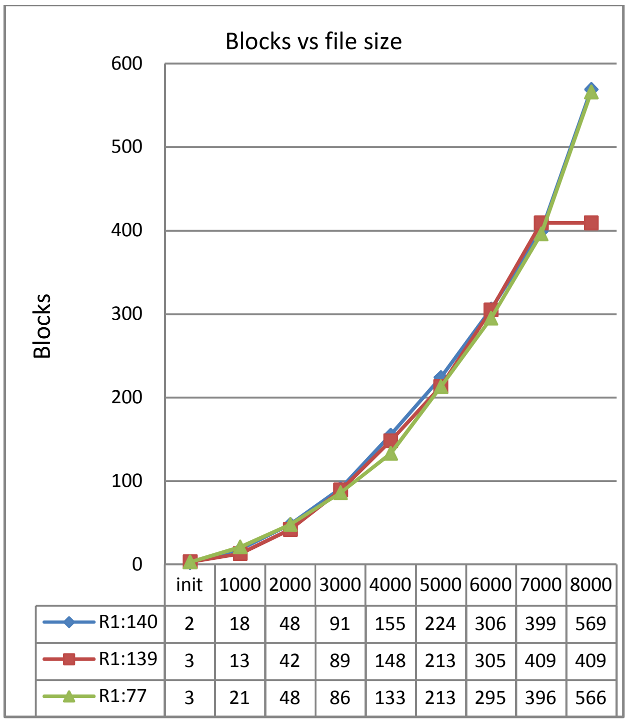
Fig. 7 reveals that the newly developed system's block placement logic distributes replicas by considering utilization of both datanode and cluster. The datanode that was previously highly utilized e.g. R1:139 does not frequently get replicas compared to those datanodes that were previously underutilized e.g. R1:77.

The interval that starts from init to 3000 on horizontal axis shows that datanode R1:139 have been chosen most frequently compared to datanode R1:77. When file size become large, from 4000 to 8000 interval on horizontal axis, the highly utilized datanode R1:139 starts to get less number of replicas compared to other datanodes R1:140 and R1:77 whose capacity was relatively higher than that of datanode R1:139. It should be noted that datanode R1:139 was not completely ignored by the system, but it (datanode R1:139) was less often selected due to its tendency of getting to an over utilized state once few number of blocks were placed on it.

Fig. 8 reveals that the original system's block placement logic distributes replicas by randomly selecting datanodes and it also put into account the remaining space of a datanode in a cluster.

The datanodes that were previously more or less utilized are not considered. The datanode (R1:139) that was previously highly utilized continues to be over utilized while datanode (R1:77) that was previously underutilized continues to be underutilized.

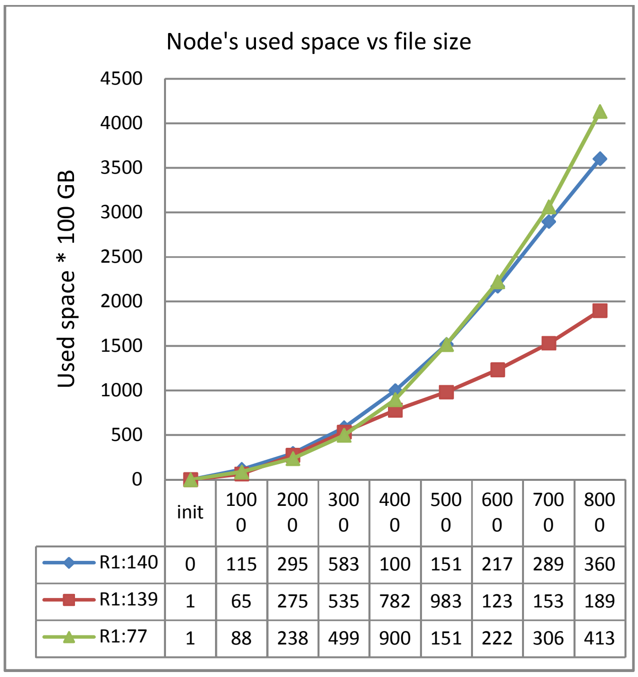
A sharp fall horizontal red line implies that datanode R1:139 have reached to the extremely over utilized point beyond which only few blocks can be placed on the datanode.



	init	1000	2000	3000	4000	5000	6000	7000	8000
R1:140	2	18	48	91	155	224	306	399	569
R1:139	3	13	42	89	148	213	305	409	409
R1:77	3	21	48	86	133	213	295	396	566

Fig. 8 Block distribution across the original Hadoop system's cluster

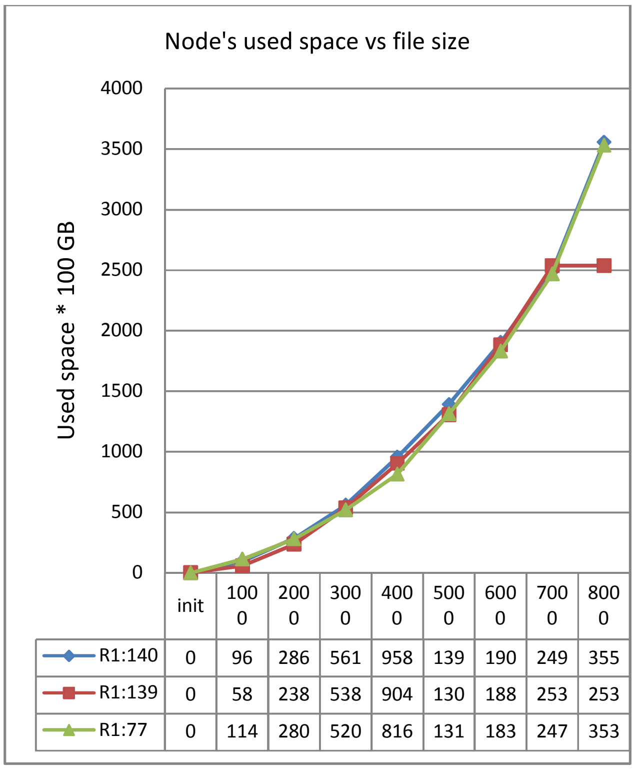
C. Used Space versus File Size



	init	1000	2000	3000	4000	5000	6000	7000	8000
R1:140	0	115	295	583	100	151	217	289	360
R1:139	1	65	275	535	782	983	123	153	189
R1:77	1	88	238	499	900	151	222	306	413

Fig. 9 Nodes used space in modified Hadoop cluster

Fig. 9 shows that the newly developed Hadoop cluster uses more space of nodes with little utilization value. Nodes with more free space are chosen most often compared to those with little free space. The newly implemented solution works in a way that do don't violate original Hadoop block placement policy.

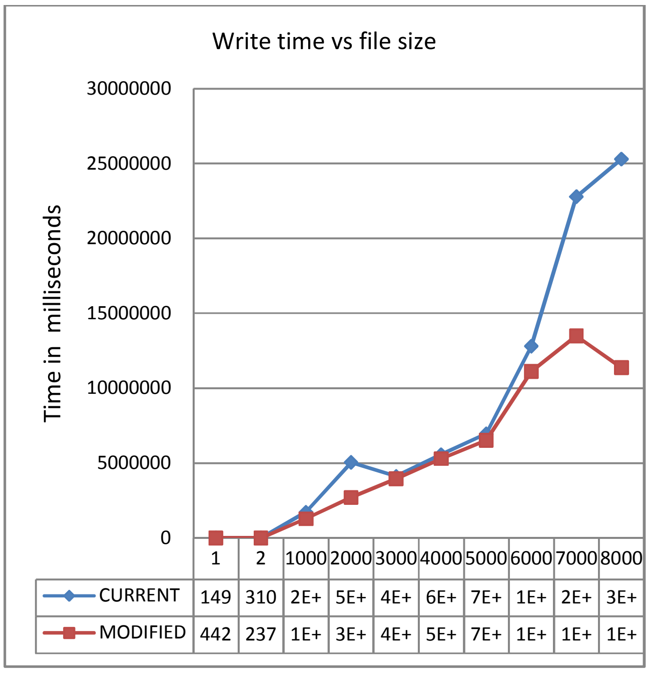


	init	100 0	200 0	300 0	400 0	500 0	600 0	700 0	800 0
R1:140	0	96	286	561	958	139	190	249	355
R1:139	0	58	238	538	904	130	188	253	253
R1:77	0	114	280	520	816	131	183	247	353

Fig. 10 Nodes used space in original Hadoop cluster

Fig. 10 above shows that, current hadoop system randomly chooses datanodes such that node's previously used space is less or even not considered. Regardless of nodes and cluster utilization, the current system has consumed approximately equal space from all datanodes. Very trivial evidence can be seen from Fig. 10 where by all datanodes' lines are approaching and crossing (with little deviation) to each other, at 3000, 5000 and 7000 points on horizontal axis. A tendency by which the lines lagging and crossing each other as it has been shown in Fig. 10 portrays that the original hadoop system do so (distributes blocks that way) to keep a little difference in datanodes' cumulative sum of used space. At point 7000 on horizontal axis it can be seen that all nodes attain approximately the same cumulative sum of used space, whose value rounds off to 2500 or 25 Gigabytes (during data collection phase every collected value for datanode's used space was multiplied by 100 for clarity reason and presentation purpose). Point above 7000 seems to be an extreme condition to datanode R1:139, red in color, since at this point it was less used compared to other datanodes, results to a relatively low cumulative sum of used space, about 2539 or 25.39 Gigabytes, compared to other datanodes whose values round off to 3500 or 35 Gigabytes.

D. Write Time Comparison



	1	2	1000	2000	3000	4000	5000	6000	7000	8000
CURRENT	149	310	2E+	5E+	4E+	6E+	7E+	1E+	2E+	3E+
MODIFIED	442	237	1E+	3E+	4E+	5E+	7E+	1E+	1E+	1E+

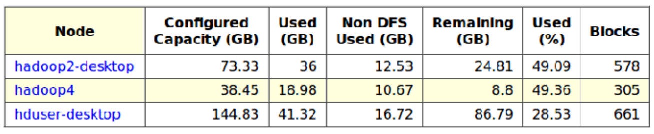
Fig. 11 Write time comparisons between new and original replication target chooser logic in HDFS system

Fig. 11 above shows that, the global time taken for a client to write blocks of data to the two file systems seem to have minor difference from each other. A sharp rise in time from 1000 to 2000 on horizontal axis, for a blue line (original system) might be due to environment factors such as network and communication traffics across the cluster.

From Fig. 11, it can be revealed that, the newly implemented Hadoop system takes less cumulative global time for an HDFS client to write blocks of data to the cluster compared to the time taken for the HDFS client to write blocks of data on datanodes in original Hadoop cluster.

E. Impact of Balancer Utility

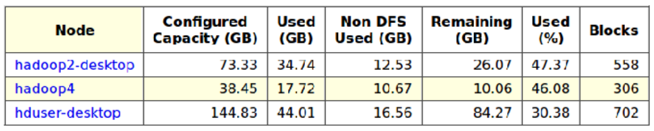
The following data in visual presentation were collected after running hadoop balancer utility on modified hadoop cluster. A bout 1544 blocks was written, as shown in Fig. 12.



Node	Configured Capacity (GB)	Used (GB)	Non DFS Used (GB)	Remaining (GB)	Used (%)	Blocks
hadoop2-desktop	73.33	36	12.53	24.81	49.09	578
hadoop4	38.45	18.98	10.67	8.8	49.36	305
hduser-desktop	144.83	41.32	16.72	86.79	28.53	661

Fig. 12 Unbalanced modified Hadoop cluster

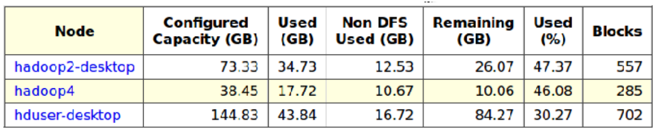
After running Hadoop balancer about 1581blocks were found, as shown in Fig. 13. It implies that about 27 blocks were over replicated by the balancer.



Node	Configured Capacity (GB)	Used (GB)	Non DFS Used (GB)	Remaining (GB)	Used (%)	Blocks
hadoop2-desktop	73.33	34.74	12.53	26.07	47.37	558
hadoop4	38.45	17.72	10.67	10.06	46.08	306
hduser-desktop	144.83	44.01	16.56	84.27	30.38	702

Fig. 13 Modified Hadoop cluster after executing the balancer

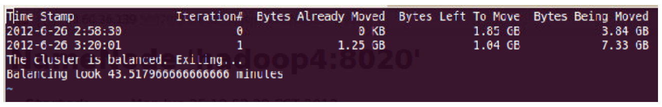
Few minutes later, a total number of 1544 blocks were found, as presented in Fig. 14. It provides vivid evidence that implies that over replicated blocks were implicitly removed.



Node	Configured Capacity (GB)	Used (GB)	Non DFS Used (GB)	Remaining (GB)	Used (%)	Blocks
hadoop2-desktop	73.33	34.73	12.53	26.07	47.37	557
hadoop4	38.45	17.72	10.67	10.06	46.08	285
hduser-desktop	144.83	43.84	16.72	84.27	30.27	702

Fig. 14 Balanced modified Hadoop system's cluster

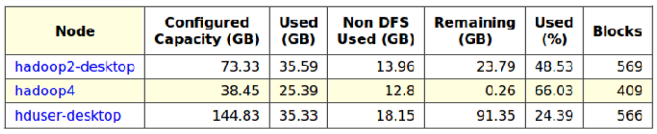
After Hadoop balancer executed completely, the time taken for execution process to complete was captured and visualized as shown in Fig. 15. It can be seen that time required to execute Hadoop balancer on the newly implemented Hadoop cluster is about 43.5180 minutes.



```
Time Stamp                Iteration#  Bytes Already Moved  Bytes Left To Move  Bytes Being Moved
2012-6-26 2:58:30                  0                 0 KB             1.85 GB            3.84 GB
2012-6-26 3:20:01                  1              1.25 GB             1.04 GB            7.33 GB
The cluster is balanced. Exiting...
Balancing took 43.51796666666666 minutes
~
```

Fig. 15 Balancer global time taken in modified Hadoop cluster

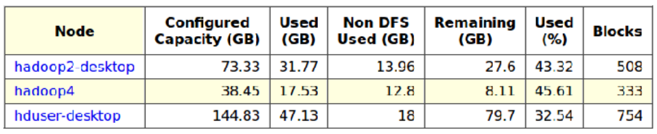
The following data in visual presentation were collected after running hadoop balancer utility on original (unmodified) Hadoop cluster. A total number of 1544 blocks were written, as shown in Fig. 16.



Node	Configured Capacity (GB)	Used (GB)	Non DFS Used (GB)	Remaining (GB)	Used (%)	Blocks
hadoop2-desktop	73.33	35.59	13.96	23.79	48.53	569
hadoop4	38.45	25.39	12.8	0.26	66.03	409
hduser-desktop	144.83	35.33	18.15	91.35	24.39	566

Fig.16 Unbalanced original Hadoop cluster

After running Hadoop balancer a total number of 1595 blocks were found, as shown in Fig. 17. It implies that about 51 blocks were over replicated by the balancer.

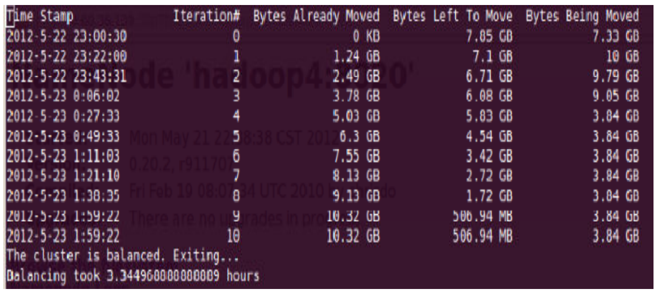


Node	Configured Capacity (GB)	Used (GB)	Non DFS Used (GB)	Remaining (GB)	Used (%)	Blocks
hadoop2-desktop	73.33	31.77	13.96	27.6	43.32	508
hadoop4	38.45	17.53	12.8	8.11	45.61	333
hduser-desktop	144.83	47.13	18	79.7	32.54	754

Fig. 17 Original Hadoop cluster after executing the balancer

The global time taken by Hadoop balancer to redistribute blocks in original Hadoop cluster was captured and presented in visual aid as shown in Fig. 18. It can be seen that about 3.3450 hours elapsed for the balancer to execute successfully.

Comparing this value to one obtained in Fig. 15, it implies that the newly (modified) implemented Hadoop cluster fairly and evenly distributes replicas of blocks to the extent that Hadoop balancer doesn't consume much time in redistributing blocks from over utilized datanodes to underutilized datanodes.



```
Time Stamp              Iteration#  Bytes Already Moved  Bytes Left To Move  Bytes Being Moved
2012-5-22 23:00:30               0                 0 KB             7.85 GB            7.33 GB
2012-5-22 23:22:00               1              1.24 GB              7.1 GB              10 GB
2012-5-22 23:43:31               2              2.49 GB             6.71 GB            9.79 GB
2012-5-23 0:06:02                3              3.78 GB             6.08 GB            9.05 GB
2012-5-23 0:27:33                4              5.03 GB             5.83 GB            3.84 GB
2012-5-23 0:49:33                5               6.3 GB             4.54 GB            3.84 GB
2012-5-23 1:11:03                6              7.55 GB             3.42 GB            3.84 GB
2012-5-23 1:21:10                7              8.13 GB             2.72 GB            3.84 GB
2012-5-23 1:38:35                8              9.13 GB             1.72 GB            3.84 GB
2012-5-23 1:59:22                9             10.32 GB           506.94 MB            3.84 GB
2012-5-23 1:59:22               10             10.32 GB           506.94 MB            3.84 GB
The cluster is balanced. Exiting...
Balancing took 3.344960000000009 hours
```

Fig. 18 Balancer output in original Hadoop cluster

With respect to Figs. 13 and 14 it can be seen that the modified system has over replicated about 27 (approximately to 1.75 percent) blocks, while Figs. 16 and 17 reveal that the original system has over replicated about 51 (approximately to 3.3 percent) blocks. Both systems' over replication states were immediately corrected by the two systems implicitly, in few minutes.

From Figs. 15 and 18 it can be seen that the modified system has been lowered the traditional balancer global execution time to 22 percent.

IV. Conclusion

The primary objectives of this research work have been successfully achieved. Our newly implemented solution fairly and evenly distributes replicas of blocks to datanodes by considering datanode and cluster utilization. Global time to execute hadoop balancer has been lowered to almost 22 percent; this value portrays the significance of our solution in time sensitive applications that take place in distributed computing world. Decline in number of re-replicated blocks from 51 to 27 after running hadoop balancer on modified hadoop cluster, indicates suitability and effectiveness of our modified system.

Further improvements to the solutions presented in this research work are needed to help the system achieve better performance with little system resources usage. To minimize delay in the system, caching of live blocks report requested by dynamic balancer from namenode is worth implemented. Information about cluster size and total load on the cluster must be cached for a reasonable amount of time so as to avoid similar requests for same information from the name node which incurs too much delay that degrades overall system performance. We reserved these features to be implemented later in next releases.

Appendix: Glossary Terms

HDFS: Hadoop Distributed File System
Replicas: Redundant copies of a block
Cluster: A network of computers formally known as Namenode and Datanodes
Balancer: Hadoop utility program used to keep cluster in a

balanced state.

Amazon EC2: Amazon Elastic Compute Cloud allows users to rent virtual computers on which to run their own computer applications.

Namenode: A computer or server that is dedicated to provide metadata services and manage hadoop cluster file system consistency.

Datanode: A computer or server that is dedicated to store data and perform cluster tasks under the governance of Namenode.

MapReduce: Hadoop framework developed in Java, which offers developers, distributed programming environment or services.

HDFS Client: Client program developed in Java or other language, which writes and reads files to and fro hadoop cluster.

R1:139: DataNode whose local IP address is 10.60.36.139 in the first rack, limited to the context of this project.

R1:140: DataNode whose local IP address is 10.60.36.140 in the first rack, limited to the context of this project.

R1:77: DataNode whose local IP address is 10.60.36.77 in the first rack, limited to the context of this project.

Original: Unmodified Hadoop system, part for the input of this project work

Current: Modified Hadoop system, the output of this project work

ACKNOWLEDGMENT

We express our sincere gratitude to School of software engineering, Tongji University and Shanghai Municipal government for financing our study in China, without which this work could not be accomplished.

REFERENCES

[1] http://www.aosabook.org/en/hdfs.html
[2] Tom White, “Hadoop: The definitive guide”, 2nd ed., O’REILLY, pp. 304.
[3] Brad Hedlund, “Understanding Hadoop Cluster and the Network”, www.bradhedlund.com

Nchimbi, Edward Pius, is currently pursuing a final academic year of Masters Degree in Software Engineering, in Tongji University, P.R China.

Academically, He has completed four years (2006-2010) Bachelor degree studies in Telecommunication Engineering in Huazhong University of Science & Technology, P.R China. In 2005/2006 he did Chinese Language programme in Shandong University, Jinan, and P.R China. From 2003 to 2005, he pursued Advanced Level studies, in Physics, Chemistry and Advanced Mathematics in Ilboru Secondary School, Arusha, Tanzania.

Technically, he had worked for SAP China Labs, as intern, from 2011 to May, 2012.